

Computabilidad

Índice

1. Programas y funciones computables	3
1.1. El lenguaje \mathcal{S}	3
1.2. Programas de \mathcal{S}	3
1.3. Macros	4
1.4. Estados y snapshots	5
1.5. Funciones computables	6
1.6. Predicados	6
2. Funciones primitivas recursivas	7
2.1. Composición	7
2.2. Recursión	7
2.3. Clases PRC	8
2.4. Predicados primitivos recursivos	8
2.5. Cuantificadores acotados	8
2.6. Minimalización	9
2.7. Codificación de pares y números de Gödel	9
3. Programas Universales	10
3.1. Codificación de programas	10
3.2. El Halting Problem	10
3.3. Teorema de Universalidad	11
3.4. Conjuntos recursivamente enumerables	12
3.5. Teorema del Parámetro	12
3.6. El teorema de la recursión	12
3.7. Teorema de Rice	13

1. Programas y funciones computables

1.1. El lenguaje \mathcal{S}

Definimos el lenguaje \mathcal{S} de la siguiente manera:

1. Las variables de entrada de \mathcal{S} son $\{X_i\}_{i \in \mathbb{N}}$.
2. La variable de retorno de \mathcal{S} es Y .
3. Las variables locales de \mathcal{S} son $\{Z_i\}_{i \in \mathbb{N}}$.
4. Las variables de \mathcal{S} toman valores en \mathbb{N}_0 .
5. Las etiquetas de \mathcal{S} son $A_1, B_1, C_1, D_1, E_1, A_2, B_2, C_2, D_2, E_2, A_3, \dots$
6. Las instrucciones de \mathcal{S} son:
 - $V \leftarrow V + 1$
 - $V \leftarrow V - 1$
 - IF $V \neq 0$ GOTO L

donde V es una variable y L es una etiqueta. Las instrucciones pueden además tener una etiqueta.

Veamos qué quiere decir cada una de estas afirmaciones y como funciona el lenguaje.

La primera afirmación dice que hay una variable de entrada por cada número natural, y que a la i -ésima variable de entrada la llamamos X_i . Un programa de \mathcal{S} que toma n variables de entrada comienza con $X_m = 0$ para todo $m > n$. La variable X_1 es comunmente denominada X .

La segunda afirmación dice que todo programa del lenguaje \mathcal{S} va a devolver el valor que tenga la variable Y al finalizar la ejecución de la última instrucción que ejecute el programa.

La tercera afirmación dice que cuando un programa de \mathcal{S} use variables que no son parte de la entrada (variables locales), estas serán una por cada natural y llamaremos Z_i a la i -ésima variable local. Por definición empiezan todas con el valor 0.

La cuarta afirmación nos dice que los valores que pueden tomar las variables son los naturales y el cero (es decir, enteros no negativos, o enteros mayores o iguales a cero). Al igual que lo que pasa con la variable X_1 , podemos llamarle Z a la variable Z_1 .

La quinta afirmación nos dice que las etiquetas que tenemos disponibles para nuestro programa serán A_i, B_i, C_i, D_i y E_i con i un número natural, que nuevamente podemos omitir si es el número 1.

La sexta y última afirmación habla de las instrucciones del lenguaje \mathcal{S} . Estas son:

- $V \leftarrow V + 1$ que incrementa el valor de la variable V en 1.
- $V \leftarrow V - 1$ que decrementa el valor de la variable V en 1 si V no vale 0. Si $V = 0$ no hace nada.
- IF $V \neq 0$ GOTO L , que ejecuta la primera instrucción del programa con etiqueta L si la variable V tiene un valor distinto de 0. Si no existe ninguna instrucción con la etiqueta L entonces el programa termina. La etiqueta E se suele utilizar para terminar un programa, y por lo tanto no se suele usar.

También vamos a agregar la instrucción $V \leftarrow V$ que no tiene ningún efecto pero nos será útil más adelante. Las instrucciones pueden además tener una etiqueta. Un ejemplo de una instrucción con una etiqueta es: $[B]X \leftarrow X + 1$

1.2. Programas de \mathcal{S}

Un programa en el lenguaje \mathcal{S} es una lista finita de instrucciones de \mathcal{S} que se ejecutan comenzando en la primera instrucción y en un orden definido de la siguiente manera siendo n la cantidad de instrucciones del programa:

- Si la i -ésima instrucción es $V \leftarrow V + 1$, $V \leftarrow V - 1$ o $V \leftarrow V$, luego de ejecutar dicha instrucción se ejecutará la $i + 1$ -ésima instrucción en caso de que esta exista. Si $i = n$ el programa termina luego de ejecutar la n -ésima instrucción.
- Si la i -ésima instrucción es $\text{IF } V \neq 0 \text{ GOTO } L$ y la variable V tiene valor 0, se procede igual que en el caso anterior.
- Si la i -ésima instrucción es $\text{IF } V \neq 0 \text{ GOTO } L$ y la variable V tiene un valor distinto de 0, y existe al menos una instrucción con etiqueta L , entonces se ejecutará, luego de la i -ésima instrucción, la primer instrucción del programa que tenga dicha etiqueta (si más de una instrucción comparten una misma etiqueta, sólo la primera será válida, y todas las demás serán ignoradas).
- Si la i -ésima instrucción es $\text{IF } V \neq 0 \text{ GOTO } L$ y la variable V tiene un valor distinto de 0, y no existe ninguna instrucción con etiqueta L , entonces el programa terminará luego de ejecutar dicha instrucción.

Un ejemplo de un programa en \mathcal{S} es

```
[A]  X ← X - 1
      Y ← Y + 1
      IF X ≠ 0 GOTO A
```

Este programa computa la función que vale 1 si $x = 0$ o x si $x \neq 0$. El siguiente programa computa la función identidad:

```
[A]  IF X ≠ 0 GOTO B
      Z ← Z + 1
      IF Z ≠ 0 GOTO E
[B]  X ← X - 1
      Y ← Y + 1
      Z ← Z + 1
      IF Z ≠ 0 GOTO A
```

Las instrucciones $Z \leftarrow Z + 1$ están para que al ejecutar la instrucción $\text{IF } Z \neq 0 \text{ GOTO } L$ (siendo $L = E$ o $L = A$), siempre se cumpla la condición. Podemos entonces, si usamos para esto una variable local que no sea usada en el resto del programa, simular la instrucción $\text{GOTO } L$. A estas instrucciones, que no vienen provistas por el lenguaje, pero podemos simularlas con dos o más instrucciones de \mathcal{S} , las denominamos macros.

1.3. Macros

Una macro es una instrucción que no viene dada por el lenguaje pero puede ser simulada por otra instrucción. Por ejemplo, la instrucción $\text{GOTO } L$ es una macro, ya que puede ser simulada por las instrucciones

```
Z ← Z + 1
IF Z ≠ 0 GOTO L
```

A las instrucciones que simulan una macro las denominamos expansión de la macro. Ya vimos un programa que computa la función identidad, por lo que en principio podríamos decir que

tenemos la macro $V \leftarrow V'$. Esto sin embargo no es así, ya que al guardar en Y el valor que tiene X con este programa, X pierde su valor y pasa a valer 0. Sin embargo podemos simular esa instrucción con la siguiente macro

```
[A]  IF  $X \neq 0$  GOTO B
      GOTO C
[B]   $X \leftarrow X - 1$ 
       $Y \leftarrow Y + 1$ 
       $Z \leftarrow Z + 1$ 
      GOTO A
[C]  IF  $Z \neq 0$  GOTO D
      GOTO E
[D]   $Z \leftarrow Z - 1$ 
       $X \leftarrow X + 1$ 
      GOTO C
```

que copia el valor de X en Y , pero también lo conserva en X . Luego podemos usar la macro $V \leftarrow V'$. No debe preocuparnos el hecho de usar variables locales o etiquetas que podamos usar luego en el programa, siempre y cuando tengamos cuidado y renombremos las variables para que no aparezcan en otra parte del programa.

Ahora podemos escribir un programa que compute la suma de dos números enteros, y gracias al uso de macros este programa será bastante corto y sencillo de entender:

```
       $Y \leftarrow X_1$ 
       $Z \leftarrow X_2$ 
[B]  IF  $Z \neq 0$  GOTO A
      GOTO E
[A]   $Z \leftarrow Z - 1$ 
       $Y \leftarrow Y + 1$ 
      GOTO B
```

1.4. Estados y snapshots

Un estado de un programa es una asignación de valores a sus variables. Un estado le asigna un valor a todas las variables que aparecen en el programa, y no le asigna más de una vez un valor a una misma variable. Si por ejemplo tenemos el programa \mathcal{P} dado por

$X \leftarrow X - 1 \quad Y \leftarrow Y + 1$

Los siguientes son estados válidos de \mathcal{P}

$$X = 3 \quad Y = 2$$

$$X = 2 \quad Y = 2 \quad Z_4 = 8$$

En el segundo caso, no importa que Z_4 no aparezca en \mathcal{P} , ya que el estado le asigna un valor a todas las variables que aparecen en \mathcal{P} .

Los siguientes estados no son válidos para \mathcal{P}

$$X = 3 \quad X = 2 \quad Y = 1$$

$$X_2 = 3 \quad Y = 1$$

El primero no lo es ya que le asigna dos veces un valor a X , y el segundo porque no le asigna ninguno, y X es una variable del programa \mathcal{P} . Para que un estado sea válido no es necesario que este sea alcanzable por el programa.

Definimos un snapshot de un programa como un par (i, σ) donde $1 \leq i \leq n + 1$ (siendo n la longitud del programa, es decir, la cantidad de instrucciones del programa) y σ un estado. El número i representa la instrucción a ejecutarse si $i \leq n$, y cuando $i = n + 1$ representa la terminación del programa. Un snapshot se dice terminal si $i = n + 1$.

Para cada snapshot no terminal, definimos su sucesor como el snapshot que se obtiene de la siguiente manera:

- Si la i -ésima instrucción es $V \leftarrow V + 1$, entonces reemplazamos i por $i + 1$ y la ecuación $V = t$ por la ecuación $V = t + 1$ en σ donde t es el valor de V en el estado σ .
- Si la i -ésima instrucción es $V \leftarrow V - 1$, entonces reemplazamos i por $i + 1$ y la ecuación $V = t$ por la ecuación $V = \max(t - 1, 0)$ en σ donde t es el valor de V en el estado σ .
- Si la i -ésima instrucción es $V \leftarrow V$, entonces reemplazamos i por $i + 1$ y no modificamos σ .
- Si la i -ésima instrucción es IF $V \neq 0$ GOTO L , en caso de que exista una instrucción con etiqueta L , reemplazamos i por el número de esa instrucción (la primera si hay varias), caso contrario reemplazamos i por $n + 1$. En ambos casos conservamos σ sin modificarlo.

Un cálculo de un programa \mathcal{P} es una lista de snapshots s_1, s_2, \dots, s_k tal que para todo $1 \leq i < k$ se tiene que s_{i+1} es el sucesor de s_i , y s_k es terminal.

1.5. Funciones computables

Un estado se dice inicial si la variable Y y todas las variables Z_i valen 0. Un snapshot $s = (i, \sigma)$ se dice inicial si $i = 1$ y σ es un estado inicial.

Si existe un cálculo de \mathcal{P} que comienza con un snapshot inicial $s = (i, \sigma)$ donde $X_1 = r_1, \dots, X_n = r_n$ y todas las demás variables están en cero en σ , entonces decimos que $y = \psi_P^{(n)}(r_1, \dots, r_n)$ si y es el valor que toma Y en el estado terminal de dicho cálculo. Caso contrario decimos que $\psi_P^{(n)}(r_1, \dots, r_n)$ está indefinido.

Dada una función g , decimos que \mathcal{P} computa g si

$$g(r_1, \dots, r_n) = \psi_P^{(n)}(r_1, \dots, r_n)$$

cuando ambos lados de la ecuación están definidos, y cuando una se indefine la otra también.

En este caso decimos que g es parcialmente computable (o parcialmente recursiva), y es total si nunca se indefine. Una función se dice computable (o recursiva) si es parcialmente computable y total.

1.6. Predicados

Decimos que $P(V_1, \dots, V_n)$ es un predicado si toma dos valores de verdad (TRUE y FALSE) a los que representaremos por 1 y 0 respectivamente. Si un predicado es computable, entonces

```
Z ← P(V1, ..., Vn)
IF Z ≠ 0 GOTO L
```

es la expansión de la macro IF $P(V_1, \dots, V_n)$ GOTO L . En este caso, $Z \leftarrow P(V_1, \dots, V_n)$ es también una macro que resulta de reemplazar Y por Z y X_i por V_i para todo $1 \leq i \leq n$ en el programa que computa $P(V_1, \dots, V_n)$.

2. Funciones primitivas recursivas

2.1. Composición

Si tenemos $f, g : \mathbb{N}_0 \rightarrow \mathbb{N}_0$, decimos que

$$h(x) = f(g(x))$$

es la composición de f con g .

Si tenemos $f; \mathbb{N}_0^k \rightarrow \mathbb{N}_0$ y $g_1, \dots, g_k : \mathbb{N}_0^n \rightarrow \mathbb{N}_0^k$ decimos que h se obtiene de f, g_1, \dots, g_k por composición si

$$h(x_1, \dots, x_n) = f(g_1(x_1, \dots, x_n), \dots, g_k(x_1, \dots, x_n))$$

h está definida por $f(z_1, \dots, z_k)$ donde $z_i = g_i(x_1, \dots, x_n)$ para todo $1 \leq i \leq k$ si z_1, \dots, z_k están definidas y $f(z_1, \dots, z_k)$ también.

Si f, g_1, \dots, g_k son (parcialmente) computables, h también lo es y se computa con el siguiente programa:

```
Z1 ← g1(X1, ..., Xn)
...
Zk ← gk(X1, ..., Xn)
Y ← f(Z1, ..., Zk)
```

2.2. Recursión

Si k es constante y

$$h(0) = k$$

$$h(t+1) = g(t, h(t))$$

para alguna función total g de dos variables, decimos que h se obtiene de g por recursión.

Si g es computable, h es computable por el siguiente programa:

```
Y ← k
[A] IF X = 0 GOTO E
    Y ← g(Z, Y)
    X ← X - 1
    GOTO A
```

Donde $Y \leftarrow k$ es la macro que consiste en k instrucciones $Y \leftarrow Y + 1$ cuando Y vale 0. Si Y no vale 0 también podemos resetearla a 0 con una macro, y $\text{IF } X = 0 \text{ GOTO } E$ se corresponde con la siguiente macro:

```
IF X ≠ 0 GOTO A
GOTO E
[A] siguiente instrucción
```

También podemos definir recursión para funciones de varias variables de la siguiente manera:

$$h(x_1, \dots, x_n, 0) = f(x_1, \dots, x_n)$$

$$h(x_1, \dots, x_n, t+1) = g(t, h(x_1, \dots, x_n, t), x_1, \dots, x_n)$$

Y también podemos ver que si f y g son computables, h es computable por el siguiente programa:

[A] $Y \leftarrow f(X_1, \dots, X_n)$
 IF $X_{n+1} = 0$ GOTO E
 $Y \leftarrow g(Z, Y, X_1, \dots, X_n)$
 $Z \leftarrow Z + 1$
 $X_{n+1} \leftarrow X_{n+1} - 1$
 GOTO A

2.3. Clases PRC

Una clase PRC (primitive recursively closed) o clase cerrada por primitivas recursivas, es una clase cerrada por recursión, composición y que contiene a las funciones iniciales, que son:

$$s(x) = x + 1$$

$$n(x) = 0$$

$$u_i^n(x_1, \dots, x_n) = x_i \quad 1 \leq i \leq n$$

La clase de funciones computables es una clase PRC. Ya vimos que las funciones iniciales son computables y que composición y recursión de computables es computable, luego las computables son una clase PRC. Una función es primitiva recursiva si puede obtenerse de las funciones iniciales mediante una cadena finita de composiciones y recursiones. Es fácil ver que toda clase PRC contiene a las funciones primitivas recursivas. A su vez, toda función que pertenece a todas las clases PRC, es primitiva recursiva, luego las primitivas recursivas constituyen una clase PRC.

De lo dicho anteriormente podemos deducir fácilmente que toda función primitiva recursiva es computable.

2.4. Predicados primitivos recursivos

Un predicado es una función $P : \mathbb{N}^n \rightarrow \{0, 1\}$ con $n \in \mathbb{N}$. La definición de predicado primitivo recursivo es análoga a la de función primitiva recursiva. Si P y Q son predicados primitivos recursivos, entonces $\neg P$ ($1 - P$), $P \wedge Q$ ($P \cdot Q$), $P \vee Q$ ($1 - (1 - P) \cdot (1 - Q)$) y $P \rightarrow Q$ ($1 - P \cdot (1 - Q)$) lo son.

El predicado

$$f(x_1, \dots, x_n) = \begin{cases} g(x_1, \dots, x_n) & \text{si } P(x_1, \dots, x_n) \\ h(x_1, \dots, x_n) & \text{caso contrario} \end{cases}$$

es computable si P es computable.

2.5. Cuantificadores acotados

Sea C una clase PRC. Si $f(t, x_1, \dots, x_n)$ pertenecen a C , entonces también pertenecen a C las funciones:

$$g(y, x_1, \dots, x_n) = \sum_{t=0}^y f(t, x_1, \dots, x_n)$$

y

$$h(y, x_1, \dots, x_n) = \prod_{t=0}^y f(t, x_1, \dots, x_n)$$

Si P es un predicado se tiene:

$$(\forall t)_{\leq y} P(t, x_1, \dots, x_n) \Leftrightarrow \left[\prod_{t=0}^y f(t, x_1, \dots, x_n) \right] = 1$$

$$(\exists t)_{\leq y} P(t, x_1, \dots, x_n) \Leftrightarrow \left[\sum_{t=0}^y f(t, x_1, \dots, x_n) \right] \neq 0$$

Luego si P es computable (o primitivo recursivo), tanto $(\forall t)_{\leq y} P(t, x_1, \dots, x_n)$ como $(\exists t)_{\leq y} P(t, x_1, \dots, x_n)$ lo son.

2.6. Minimalización

Si $P(t, x_1, \dots, x_n)$ pertenece a una clase PRC C , entonces también pertenece a C la función

$$g(y, x_1, \dots, x_n) = \sum_{u=0}^y \prod_{t=0}^u \alpha(P(t, x_1, \dots, x_n))$$

La productoria da 1 sii para todo $0 \leq t \leq u$ el predicado $P(t, x_1, \dots, x_n)$ es falso, luego, si existe un $1 \leq u \leq y$ tal que $P(u, x_1, \dots, x_n)$ es verdadero, entonces $g(y, x_1, \dots, x_n)$ es el mínimo u que cumple dicha condición. Estamos en condiciones de definir entonces la siguiente función que también pertenece a C

$$\min_{t \leq y} P(t, x_1, \dots, x_n) = \begin{cases} g(x_1, \dots, x_n) & \text{si } (\exists t)_{\leq y} P(t, x_1, \dots, x_n) \\ 0 & \text{caso contrario} \end{cases}$$

que nos devuelve el menor entero t tal que $P(t, x_1, \dots, x_n)$ es verdadero, si dicho t es menor o igual a y , y 0 si no existe ese valor o es mayor estricto a y .

Si $P(x_1, \dots, x_n, y)$ es computable entonces también lo es

$$g(x_1, \dots, x_n) = \min_y P(x_1, \dots, x_n, y)$$

y el programa que lo computa es

```
[A]  IF  $P(X_1, \dots, X_n, Y)$  GOTO  $E$ 
       $Y \leftarrow Y + 1$ 
      GOTO  $A$ 
```

2.7. Codificación de pares y números de Gödel

Dados dos enteros x, y existe un único entero $\langle x, y \rangle$ tal que

$$\langle x, y \rangle = 2^x(2y + 1) - 1$$

Además, esta función es biyectiva y computable. Dado un número $z = \langle x, y \rangle$ decimos que $l(z) = x$ y $r(z) = y$, que son ambas funciones computables:

$$l(z) = \min_{x \leq z} [(\exists y)_{\leq z} (z = \langle x, y \rangle)]$$

$$r(z) = \min_{y \leq z} [(\exists x)_{\leq z} (z = \langle x, y \rangle)]$$

ya que claramente $x, y \leq z$.

Dada una n -upla de números enteros (a_1, \dots, a_n) le llamamos número de Gödel al número

$$[a_1, \dots, a_n] = \prod_{i=1}^n p_i^{a_i}$$

Por ejemplo $[2, 0, 3] = 2^2 * 3^0 * 5^3 = 500$.

3. Programas Universales

3.1. Codificación de programas

Usando la codificación de pares, y los números de Gödel, queremos asignarle un número a cada programa. El primer paso será asignarle un número a cada instrucción, y luego, usando los números de las instrucciones del programa, le asignaremos un número de programa.

Antes de darle un número a cada instrucción, ordenaremos las variables y las etiquetas. El orden de las variables será

$$Y, X_1, Z_1, X_2, Z_2, X_3, Z_3, \dots$$

mientras que el orden de las etiquetas será

$$A_1, B_1, C_1, D_1, E_1, A_2, B_2, C_2, D_2, E_2, A_3, \dots$$

empezando desde 1, es decir la variable Y es la variable 1, la variable X_i es la variable $2i$ y la variable Z_i es la variable $2i + 1$, mientras que para las etiquetas, la etiqueta A_i es la etiqueta $5i - 4$, la etiqueta B_i es la $5i - 3$, la etiqueta C_i tiene número $5i - 2$, la etiqueta D_i será la $5i - 1$, y por último la etiqueta E_i será la etiqueta que lleve el número $5i$. Dada una instrucción I la codificaremos con el siguiente número

$$\#(I) = \langle a, \langle b, c \rangle \rangle$$

Donde a es 0 si la instrucción no tiene etiqueta, o si tiene etiqueta, el número de la etiqueta que tenga, c es el número de la variable mencionada en la instrucción menos uno (es decir, si la instrucción menciona la variable Y , entonces $c = 0$), y b indica qué tipo de instrucción es según las siguientes reglas:

- Si la instrucción es del tipo $V \leftarrow V$ entonces $b = 0$
- Si la instrucción es del tipo $V \leftarrow V + 1$ entonces $b = 1$
- Si la instrucción es del tipo $V \leftarrow V - 1$ entonces $b = 2$
- Si la instrucción es del tipo IF $V \neq 0$ GOTO L entonces $b = \#(L) + 2$

Ahora que definimos una biyección entre \mathbb{N}_0 y las instrucciones de \mathcal{S} , podemos codificar un programa \mathcal{P} que tiene n instrucciones de la siguiente manera:

$$\#(P) = [\#(I_1), \dots, \#(I_n)] - 1$$

donde I_i es la i -ésima instrucción de \mathcal{P} . Notemos que la instrucción $Y \leftarrow Y$ tiene número 0 y por lo tanto prohibimos esta instrucción como última instrucción de un programa para evitar ambigüedades.

3.2. El Halting Problem

Definimos $\text{HALT}(x, y)$ como verdadero si $\psi_P^{(1)}(x)$ está definida, y falso si no lo está. $\text{HALT}(x, y)$ nos dice si el programa de número y termina con valor de entrada x . Para esto consideramos que y toma un sólo valor de entrada, y todos los demás son cero.

$\text{HALT}(x, y)$ no es un predicado computable. Supongamos que $\text{HALT}(x, y)$ fuese computable y veamos el siguiente programa \mathcal{P} :

$$[A] \quad \text{IF } \text{HALT}(X, X) \text{ GOTO } A$$

Si $\#(P) = y_0$ entonces si $\text{HALT}(y_0, y_0)$ termina, luego $\text{HALT}(y_0, y_0)$ no termina, pero si $\text{HALT}(y_0, y_0)$ no termina, luego $\text{HALT}(y_0, y_0)$ termina. Esta contradicción proviene de suponer que $\text{HALT}(x, y)$ es una función computable.

3.3. Teorema de Universalidad

Sea \mathcal{P} un programa y sea $\#(P) = y$. Podemos definir la función

$$\Phi^{(n)}(x_1, \dots, x_n, y) = \psi_P^{(n)}(x_1, \dots, x_n)$$

Veamos que $\Phi^{(n)}$ es una función parcialmente computable.

Si llamamos p_i al i -ésimo primo, sabemos que $p_1 = 2$, y veamos como podemos obtener el i -ésimo primo:

```

Y ← Y + 1
Y ← Y + 1
Z ← Z + 1
[A] X ← X - 1
    IF  $X_1 = 0$  GOTO E
    Z ← Y · Z
    Y ←  $\min_{y \leq z} [(\forall x)_{\leq y} \text{coprimo}(y, x) \vee x = y]$ 
    GOTO A

```

Este programa obtiene en cada paso al i -ésimo primo en Y , y el producto de los primeros i primos en Z , ya que si al producto de los primeros i primos, le restamos 1, entonces obtenemos un número que no es divisible por ninguno de esos primos, y luego hay un primo más que es menor que ese producto.

Nos falta probar que coprimo es una función computable, pero es fácil probar que el algoritmo de Euclides es computable.

Veamos ahora que $\Phi^{(n)}$ es computable por el siguiente programa (donde $Lt(Z)$ es la cantidad de instrucciones del programa P , que se computa dividiendo el número del programa por los primos en orden creciente, hasta que llegamos a 1:

```

Z ←  $X_{n+1} + 1$ 
S ←  $\prod_{i=1}^n (p_{2i})^{X_i}$ 
K ← 1
[C] IF  $K = Lt(Z) + 1 \vee K = 0$  GOTO F
    U ←  $r((Z)_K)$ 
    P ←  $P_{r(U)+1}$ 
    IF  $l(U) = 0$  GOTO N
    IF  $l(U) = 1$  GOTO A
    IF  $\text{coprimo}(P, S)$  GOTO N
    IF  $l(U) = 2$  GOTO M
    K ←  $\min_{i \leq Lt(Z)} [l((Z)_i) + 2 = l(U)]$ 
    GOTO C
[M] S ← S/P
    GOTO N
[A] S ← S · P
[N] K ← K + 1
    GOTO C
[F] Y ←  $(S)_1$ 

```

Este programa se llama programa universal. El Step-

Counter Theorem dice que $STP^{(n)}(x_1, \dots, x_n, y, t)$ dice si $\Phi(x_1, \dots, x_n, y)$ termina en a lo sumo t pasos y el algoritmo es el mismo agregándole un contador, que al llegar a t devuelve 0, y si el programa de número y termina, devuelve 1. Esta función es totalmente computable.

3.4. Conjuntos recursivamente enumerables

Un conjunto computable es el conjunto de valores de entrada para los cuales un predicado computable unario P es verdadero, y análogamente definimos conjunto primitivo recursivo en función de predicados primitivos recursivos unarios.

Un conjunto B es recursivamente enumerable si existe una función parcialmente computable g que está definida en x si $x \in B$. Un conjunto es computable si y sólo si B y su complemento son recursivamente enumerable (el algoritmo es tener una variable T que va creciendo e ir computando los STP de T pasos con las funciones que están definidas en cada uno de los dos conjuntos).

Definimos el conjunto K como el conjunto de los n tales que $HALT(n, n)$ es 1. K es recursivamente enumerable ya que $HALT$ es parcialmente computable por $\Phi^{(1)}(n, n)$ (existe un programa con un sólo parámetro que consiste en copiar al principio en X_2 el valor de X_1 , pero K no es computable. La demostración utiliza los conjuntos W_i que están definidos como los valores de entrada que hacen que el i -ésimo programa esté definido, y que son todos los conjuntos recursivamente enumerables, junto con el hecho de que si K y su complemento son recursivamente enumerables entonces K es computable.

3.5. Teorema del Parámetro

Para cada par de enteros positivos n, m existe una función $S_m^n(u_1, \dots, u_n, y)$ tal que

$$\Phi^{(m+n)}(x_1, \dots, x_m, u_1, \dots, u_n, y) = \Phi^{(m)}(x_1, \dots, x_m, S_m^n(u_1, \dots, u_n, y))$$

Para demostrar esto lo hacemos por inducción ya que

$$S_m^{k+1}(u_1, \dots, u_k, u_{k+1}, y) = S_m^k(u_1, \dots, u_k, S_{m+k}^1(u_{k+1}, y))$$

ya que S_a^b es el programa que setea las variables $a + 1, \dots, a + b$ con sus parámetros y luego corre el programa y , así, alcanza con demostrar que S_m^1 es primitiva recursiva para todo m , luego S_m^k es primitiva recursiva para todo par m, k . Para probar esto, agregamos u instrucciones $X_{m+1} \leftarrow X_{m+1} + 1$ y podemos calcular fácilmente el programa de la siguiente manera:

$$S_m^1(u, y) = \left[\left(\prod_{i=1}^u p_i \right)^{16m+10} \cdot \prod_{j=1}^{Lt(y+1)} p_{u+j}^{(y+1)_j} \right] - 1$$

Notemos que $16m + 10 = \langle 0, \langle 1, 2m + 1 \rangle \rangle$.

3.6. El teorema de la recursión

Definimos

$$\Phi_e^{(m)}(x_1, \dots, x_m) = \Phi^{(m)}(x_1, \dots, x_m, e)$$

Si $g(z, x_1, \dots, x_m)$ es una función parcialmente computable, entonces existe un e tal que

$$\Phi_e^{(m)}(x_1, \dots, x_m) = g(e, x_1, \dots, x_m)$$

para todo $(x_1, \dots, x_m) \in \mathbb{N}_0^m$. Para probar esto usamos el teorema del parámetro, y como S_m^1 es parcialmente computable entonces existe z_0 tal que

$$g(S_m^1(v, v), x_1, \dots, x_m) = \Phi^{(m+1)}(x_1, \dots, x_m, v, z_0) = \Phi^{(m)}(x_1, \dots, x_m, S_m^1(v, z_0))$$

Si tomamos $v = z_0$ y $e = S_m^1(z_0, z_0)$ entonces

$$g(e, x_1, \dots, x_m) = \Phi^{(m)}(x_1, \dots, x_m, e) = \Phi_e^{(m)}(x_1, \dots, x_m)$$

El teorema del punto fijo dice que si $f(z)$ es computable entonces hay un e tal que

$$\Phi_{f(e)}(x) = \Phi_e(x)$$

y se obtiene tomando $g(z, x) = \Phi_{f(z)}(x)$

3.7. Teorema de Rice

Sea Γ una colección de funciones parcialmente computables de una variable. Sean f, g funciones parcialmente computables de una variable tales que $f \in \Gamma, g \notin \Gamma$, el teorema de Rice dice que R_Γ , el conjunto de valores de t tales que $\Phi_t \in \Gamma$ no es computable.

Sea $P_\Gamma(t)$ la función característica de R_Γ y

$$h(t, x) = \begin{cases} g(x) & \text{si } P_\Gamma(t) = 1 \\ f(x) & \text{si } P_\Gamma(t) = 0 \end{cases}$$

Como h es parcialmente computable entonces existe un e tal que

$$\Phi_e(x) = h(e, x) = \begin{cases} g(x) & \text{si } \Phi_e \text{ pertenece a } \Gamma \\ f(x) & \text{caso contrario} \end{cases}$$

Pero entonces podemos demostrar que $e \in \Gamma$ y también $e \notin \Gamma$